Post Register Allocation Spill Code Optimization

Christopher Lupo, Kent D. Wilken
Department of Electrical and Computer Engineering
University of California, Davis, CA 95616
{lupo,wilken}@ece.ucdavis.edu

Abstract

A highly optimized register allocator should provide an efficient placement of save/restore code for procedures that contain calls. This paper presents a new approach to placing callee-saved save and restore instructions that generalizes Chow's shrink-wrapping technique[6]. An efficient, profile-guided, hierarchical spill code placement algorithm is used to analyze the structure of a procedure to calculate the minimum dynamic execution count locations to place callee-saved save and restore code. The algorithm is implemented in the Gnu Compiler Collection and has been tested on the SPEC CPU2000 Integer Benchmark suite. Results show that the technique reduces the number of dynamic load and store instructions by 15% compared to saving and restoring at procedure entry and exit while Chow's shrink-wrapping technique reduces dynamic load and store instructions by only 1% compared to saving and restoring at procedure entry and exit. The dynamic number of callee-saved save and restore instructions inserted with this new approach is never greater than the number produced by Chow's shrink-wrapping technique or the placement at procedure entry and exit.

1. Introduction

Register usage conventions are used to divide a register set into two subsets. *Callee-saved* registers are registers whose values are preserved across procedure calls, and *caller-saved* registers are registers whose values are not preserved across procedure calls. The register allocator must ensure the register usage convention is not violated by any register allocation. One aspect of ensuring a valid register allocation is to place save and restore code at valid locations for the callee-saved registers allocated in a procedure. A *save* instruction is a store to memory of a callee-saved register value prior to the callee-saved register being allocated to a variable. A *restore* instruction is a load from memory of a callee-saved register value that restores the original value to the callee-saved register. A valid placement of save and restore code for callee-saved registers ensures that a calling procedure has the same value in a callee-saved register before and after a procedure call.

This paper describes a profile guided, hierarchical spill code placement algorithm for optimizing the placement of save/restore instructions within a procedure after register allocation has been performed. The algorithm is simple, efficient, produces excellent results compared to previous techniques, and is compatible with any register allocator.

The organization of this paper is as follows. The problem being targeted in this paper is defined in Section 2. Related work is presented in Section 3. The profile guided hierarchical spill code placement algorithm is described in Section 4. The experimental method and results are given in Section 5.

2. Problem Definition

A register allocator allocates the limited set of registers available in a processor to program variables. When there are not enough registers available for all variables that need to be allocated, the register allocator must *spill* a variable. When a variable is spilled, it is temporarily allocated to memory rather than a register. The instructions used to spill a variable are store and load instructions and are referred to as *spill code*. When a register allocator utilizes callee-saved registers in an allocation, the original value in the callee-saved register is treated as a variable that must be spilled. The original value in the callee-saved register must be saved to memory before the register is used for another variable, and the original value must be restored to the same callee-saved register prior to the exit of the procedure.

Spill code instructions introduced by a register allocator are considered overhead instructions, because they are not included in the original set of instructions in the program. There is a cost associated with overhead instructions. Each overhead instruction has a dynamic execution count determined by profiling data. In the context of this paper, dynamic execution count, dynamic overhead, and cost are used interchangeably. There is also a static overhead associated with each spill code instruction inserted. Static overhead reduction is not a goal of the algorithm presented in this paper.

The problem considered in this paper is the minimum cost placement of callee-saved save and restore code after register allocation has been completed. A placement of save/restore instructions must satisfy the conditions that the register allocation is not altered and that each callee-saved register's value is consistent at procedure entry and exit.

3. Related Work

When allocating registers to variables, a register allocator may allocate a callee-saved register to a variable that spans a set of call locations, to prevent having to save and restore a caller-saved register around each call site in the set. There will be spill code overhead associated with the necessary save and restore instructions for the callee-saved registers allocated in a procedure.

One placement of callee-saved save and restore instructions that is always valid is to insert save instructions at procedure entry and restore instructions at each procedure exit. This entry/exit placement technique results in low static overhead. However, save and restore instructions placed at procedure entry/exit may be executed more often than necessary. There may be many different execution paths through a procedure, not all of them taken for each invocation. As a result, dynamic overhead for save and restore code may be suboptimal with a procedure entry/exit placement. To reduce dynamic overhead associated with placement of callee-saved save and restore instructions, Chow proposed a technique that uses data flow analysis to improve the placement of callee-saved save and restore instructions[6]. Chow's technique, called *shrink-wrapping*, identifies the regions of a procedure where a callee-saved register is allocated, and places save and restore instructions around each disjoint region of allocation. His shrink-wrapping technique places callee-saved save and restore code only on execution paths that cross regions of the procedure where a callee-saved register is allocated. Chow's iterative data flow implementation is done efficiently in a single compilation pass, and may result in code that has lower dynamic overhead than the same procedure where save and restore instructions are placed at procedure entry and exit.

Chow's shrink-wrapping technique is limited because it cannot leverage profiling data. One example where the callee-saved spill code locations computed using the shrink-wrapping technique may or may not produce lower dynamic overhead is described by Chow and is shown in Figure 1.

In the example shown in Figure 1, the placement of save and restore instructions computed using the shrink-wrapping technique will have lower dynamic overhead than

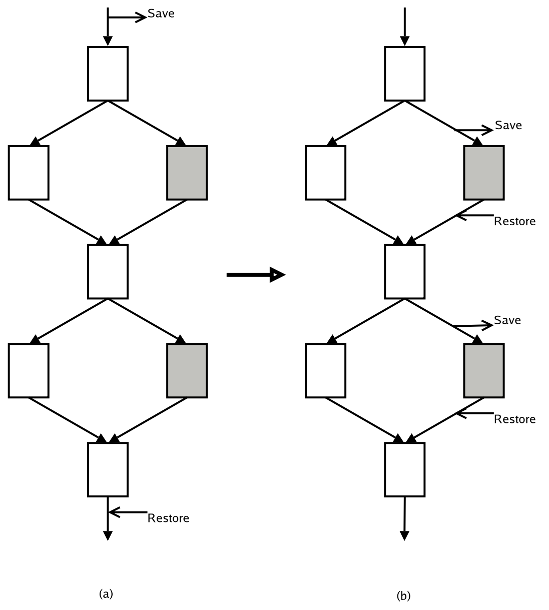

Figure 1. Example control flow graph with shaded basic blocks indicating regions of allocation for a callee-saved register where save and restore instructions are placed (a) at procedure entry and exit and (b) at locations determined using the shrink-wrapping technique.

the placement at procedure entry and exit only if the average dynamic execution count of the two basic blocks with a callee-saved register allocated is less than the dynamic execution count of the procedure entry and exit. Only when profiling data are available can the minimum dynamic overhead spill code locations be selected.

The algorithm presented in the next section is a new approach to placing save and restore instructions that generalizes Chow's shrink-wrapping technique and minimizes the dynamic overhead of callee-saved save and restore instructions. Section 4 describes an efficient, hierarchical spill code placement algorithm for computing the minimum cost locations of callee-saved save and restore instructions over the entire control flow graph for all callee-saved registers allocated in a procedure. The new approach is guaranteed to compute locations for save and restore instructions that never have greater dynamic overhead than locations determined by the shrink-wrapping technique or by the procedure entry/exit spill code placement technique, and very often have significantly lower dynamic overhead.

Like the shrink-wrapping technique, the hierarchical spill code placement algorithm presented in this paper is performed after register allocation, and is compatible with any register allocator, such as [2, 3, 4, 5, 7, 11].

4. Hierarchical Spill Code Placement

This section presents the new approach for finding a minimum dynamic execution count placement of callee-saved save and restore code.

4.1. Motivating Example

Consider the example control flow graph shown in Figure 2(a). The shaded basic blocks indicate areas where a callee-saved register has been allocated. Chow's shrink-wrapping technique will place spill instructions as shown in Figure 2(b). Save instructions are placed before basic blocks C, G, K and N, and restore instructions are placed after basic blocks F, G, K and N. The dynamic overhead of any spill code placement is computed by summing each inserted instruction's dynamic execution count. In the placement determined by the shrink-wrapping technique, the dynamic spill code overhead is 250 instruction cycles.

Another valid spill code placement for the example in Figure 2(a) is to place a save instruction in the entry basic block A and a restore instruction in the exit basic block P. The dynamic spill code overhead for the procedure entry/exit spill code placement is 200 instruction cycles. For this example, the shrink-wrapping technique produces spill code with greater dynamic overhead than the procedure entry/exit spill code placement technique. Other valid callee-saved spill code placements exist for the program example shown in Figure 2, and will be discussed later in the section.

As the size and complexity of the control flow graph increases, so do the number of valid locations for save and restore code. Comparing the cost of each set of possible save/restore locations to the set of locations determined by the shrink-wrapping technique is not only computationally expensive, but is also not precise in all cases. Consider again the example shown in Figure 2(a). If a save instruction is inserted before basic block I, and a restore instruction is inserted after basic block O, then the save/restore instruction pairs around basic blocks K and N are not necessary. Adding save and restore instructions at certain points in a program's control flow graph may allow other save and restore locations to be removed, potentially reducing dynamic spill code overhead.

4.2. Shrink-wrapping

To identify the set of locations determined by the shrink-wrapping technique, and as a starting point for the spill code placement algorithm, the data flow analysis described by Chow is used with two modifications. For the duration of this section, spill locations determined using the shrink-wrapping technique will include these modifications.

The first modification deals with loops within the control flow graph. Chow has the correct observation that save and restore instructions should, in general, not be placed inside loops because of their higher execution frequency. To handle this situation, Chow propagates artificial data flow throughout loop bodies in the control flow graph to prevent save and restore instructions from being placed inside loops. In the algorithm presented in this section, artificial data flow propagation is not necessary, as a precise, minimum cost placement of save and restore locations will be found in the control flow graph of the procedure, naturally avoiding placement of saves and restores within loops.

The second modification involves the placement of spill code on *jump edges*. A jump edge is an edge initiated by a control flow instruction whose target is not the next sequential instruction in the program. Chow's data flow analysis will identify jump edges where spill code should be placed, but Chow specifically prohibits spill code instructions from being inserted onto jump edges. Instead, when the shrink-wrapping technique identifies a jump edge where spill code should be placed, Chow propagates artificial data flow along that jump edge, and reiterates the data flow analysis. This iteration is repeated until no spill code is placed on a jump edge. For the algorithm presented in this section, no artificial data flow is propagated, and spill code instructions can be inserted on jump edges. More analysis of jump edges with spill code is given later in the section.

4.3. Program Structure

To determine a minimum cost spill code placement, it is necessary to identify a set of sufficient locations in a procedure where it is valid to place save and restore instructions. The *program structure tree* (PST) is a hierarchical representation of program structure based on *single entry single exit* (SESE) regions described by Johnson, Pearson and Pingali[8]. The PST is defined and unique for all control flow graphs. There is one difference between the PST described by Johnson et al. and the PST used in the algorithm described in this paper. Johnson et al. identify *canonical* SESE regions, where canonical indicates the smallest SESE region. The algorithm in this paper uses *maximal* SESE regions.

Definition: A SESE region (a, b) is *maximal* provided

- b post-dominates b' for any SESE region (a, b'), and
- a dominates a' for any SESE region (a', b).

The entry and exit of a SESE region are called the *boundaries* of the region. Boundaries of maximal SESE regions

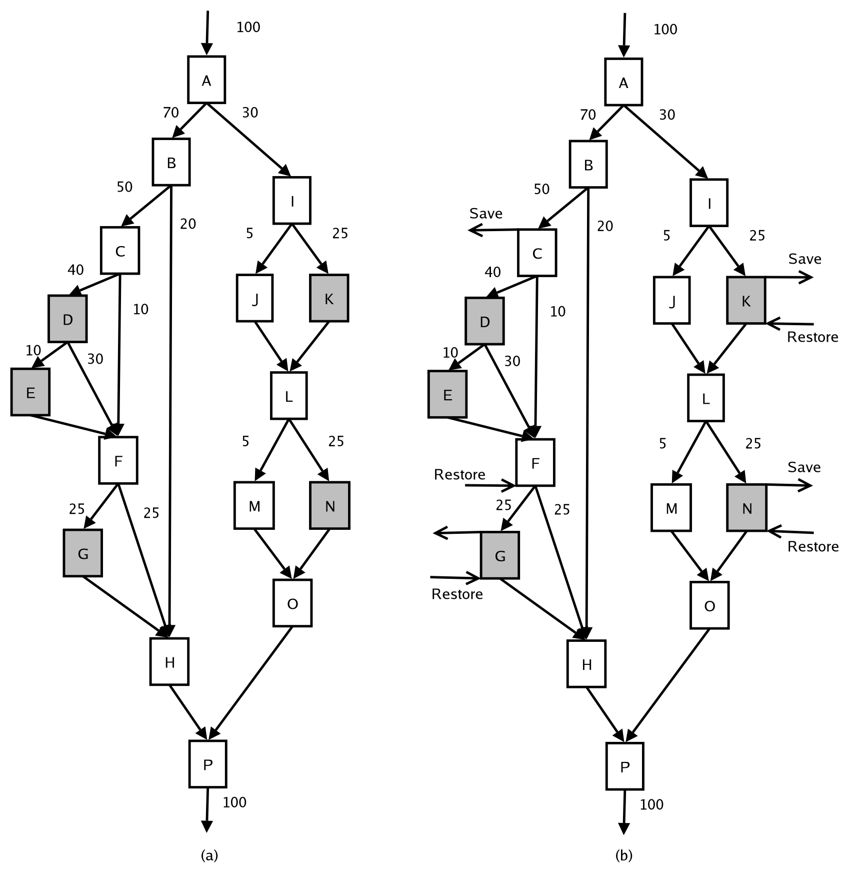

Figure 2. Motivating example for hierarchical spill code placement. (a) Shaded basic blocks indicate callee-saved register usage. Numbers next to edges indicate execution counts determined by profiling. (b) Placement of callee-saved spill code using Chow's shrink-wrapping technique.

contained within the PST, as well as save and restore locations determined by the shrink-wrapping technique, are a sufficient set of valid locations for determining a minimum cost placement of save and restore instructions in a procedure. Save and restore locations determined by the shrink-wrapping technique ensure that any callee-saved value that is saved will be restored along all execution paths from the save instruction to the exit of the procedure. Similarly, all callee-saved values restored will have been saved along all execution paths from the procedure entry to the restore instruction. The same is true for maximal SESE regions. Any value that is saved at the entry of a maximal SESE region and restored at the exit of that maximal SESE region is guaranteed to be a valid save/restore placement. The locations are sufficient for a minimum cost spill code placement because all locations within the procedure that have a change in execution frequency correspond either to the entry and exit points of maximal SESE regions, or the locations computed using the shrink-wrapping technique.

The PST is hierarchical in that the top level of the hierarchy (root node) represents the entire procedure, and the lowest levels of the hierarchy (leaf nodes) represent individual basic blocks. Each node in the PST has a parent node and a list of child nodes.

4.4. Save/Restore Sets

Save and restore locations are grouped into *save/restore sets*. A save/restore set is a collection of save and restore locations that are dependent on each other for a valid callee-saved spill code placement, and that are independent of any other save or restore locations. The save and restore locations determined by the shrink-wrapping technique are the initial save/restore sets.

The initial save/restore sets are identified using standard compiler data flow techniques for computing *variable live ranges*[1] or *webs*[10]. Save instructions represent the beginning of a web rather than definitions, and restore instructions represent the termination of a web rather than last-uses. The data flow analysis is otherwise identical.

4.5. Hierarchical Analysis

Save and restore locations have a hierarchical nature. Procedure entry and exit locations are at the top of the hierarchy, and locations determined by the shrink-wrapping technique are at the bottom of the hierarchy. For each maximal SESE region within the PST, a list of all the save/restore sets contained in the region and the total cost of those sets is computed. Analysis begins at the bottom of the hierarchy. A topological-order traversal of the PST is performed until the smallest maximal SESE region that contains at least one save/restore set is encountered. If the cost of placing save and restore locations at the boundaries of the current maximal SESE region is less than or equal to the total cost of the contained save/restore set(s), the set(s) are removed from the region, and a new save/restore set with save/restore locations at the maximal SESE region's boundaries is added. This substitution is then propagated upward through the hierarchy; deleting the cost of the removed set(s), adding the cost of the new set, and including the new set in the list of save/restore sets contained by the parent regions. If the cost of placing save and restore locations at the boundaries of the current region is not less than or equal to the total cost of the contained save/restore set(s), no changes are made. After analyzing the current region, the algorithm continues to the next region in the topological-order traversal of the PST.

The topological-order traversal of the PST guarantees that when a maximal SESE region is analyzed, all of that region's children have already been analyzed, and all save/restore sets contained within the region are at their minimum cost locations relative to the level of the hierarchy that is currently being analyzed. The final comparison in the analysis is between the cost of placing save and restore locations at procedure entry and exit to the cost of all the contained save/restore sets in the procedure, which have potentially been modified from the original locations determined by the shrink-wrapping technique.

4.6. Algorithm Details

The hierarchical spill code placement algorithm is shown below.

```
HIERARCHICAL-SPILL-CODE-PLACEMENT
1 compute PST
2 compute shrink-wrapping save/restore locations
3 compute initial save/restore sets
4 traverse PST regions in topological order
5     for each callee-saved register allocated
6         if ($Cost(maximal\ SESE\ region\ bounderies) \leq$
          $Cost(contained\ save/restore\ sets)$)
7             Remove contained save/restore sets from region
8             Create new save/restore set at region boundaries
9             Propagate changes upward through hierarchy
```

There is a cost function used in the hierarchical spill code placement algorithm to determine whether a new save/restore set should be created at the boundaries of a maximal SESE region. Two different cost models are discussed next.

4.7. Execution Count Cost Model

An execution count cost model for the hierarchical spill code placement algorithm is to have each inserted save and restore instruction weighted by the dynamic execution count of the control flow edge the instruction is inserted into. The cost of the boundaries of a maximal SESE region is the execution count of the entry edge plus the execution count of the exit edge. The hierarchical spill code placement algorithm solves the post register allocation spill code placement problem optimally with the execution count cost model.

Optimality is guaranteed using the execution count cost model because all valid save and restore locations in the procedure where dynamic execution count may change are examined in the topological-order traversal of the PST. The boundaries of each maximal SESE region represent a portion of the procedure where dynamic execution count may change. The initial placement of save and restore locations determined from the shrink-wrapping technique guarantees that no save and restore locations can be placed any closer to the portions of the procedure where each callee-saved register is allocated. The topological-order traversal evaluates each save/restore location beginning at the locations determined using shrink-wrapping, and moving hierarchically upward in the PST at each point where dynamic execution count can change. The order of the PST traversal is important, as the topological ordering ensures that for any maximal SESE region, all valid save/restore locations contained in that region have been evaluated to determine where the minimum cost locations are.

To illustrate how the hierarchical spill code placement algorithm works, consider the following example. The example program shown in Figure 2 is reproduced in Figure 3 with maximal SESE regions and initial save/restore sets identified and labeled. The save and restore locations determined by the shrink-wrapping technique are omitted from the example for clarity, but exist wherever a save/restore set intersects a control flow edge.

The traversal of the PST in topological-order for the example in Figure 3 will stop first at Region 1 because there are no smaller maximal SESE regions that contain

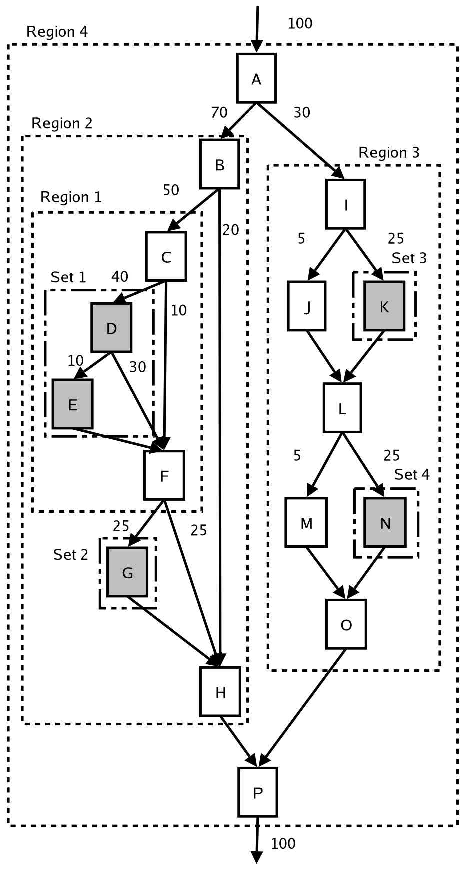

Figure 3. Example program for hierarchical spill code placement algorithm. Maximal SESE regions and save/restore sets are shown.

save/restore sets. Region 1 contains one save/restore set, Set 1. Using the execution count cost model, the cost of Set 1 is 80. The cost of the boundaries of Region 1 is 100. Since the total cost of the contained save/restore sets is less than the cost of the boundaries of the maximal SESE region, no changes are made and the topological-order traversal of the PST continues to Region 2. Region 2 contains two save/restore sets, Set 1 and Set 2. Using the execution count cost model, the cost of Set 1 is 80 and the cost of Set 2 is 50, for a combined cost of 130. The cost of the boundaries of Region 2 is 140. Since the cost of the contained save/restore sets is less than the cost of the region boundaries, no change is made and the topological-order traversal of the PST continues to Region 3. Region 3 contains two save/restore sets, Set 3 and Set 4. Using the execution count cost model, the cost of Set 3 is 50 and the cost of Set 4 is 50. The cost of the boundaries of Region 2 is 60. Since the total cost of the contained save/restore sets is greater than the cost of the boundaries of Region 3, Set 3 and Set 4 are removed, and a new save/restore set, Set 5, is created at the boundaries of Region 3. The topological-order traversal of the PST then analyzes Region 4, the top of the hierarchy. Region 4 contains three save/restore sets, Set 1, Set 2 and Set 5. The total cost of the contained save/restore sets using the execution count cost model is 190. The cost of the boundaries of Region 4 is 200. Since the cost of the contained save/restore sets is less than the cost of the maximal SESE region boundaries, no further changes are made, and the callee-saved spill code placement is optimal for this example. The final callee-saved spill code placement determined using the hierarchical spill code placement algorithm with the execution count cost model is shown in Figure 4(a).

4.8. Jump Edge Cost Model

The execution count cost model does not always accurately represent how spill code is inserted into real programs. Spill code must be inserted into a basic block, and cannot reside on a control flow edge. For example, consider save/restore set Set 1 shown in Figure 4(a). The save instruction in Set 1 is inserted into basic block D prior to the other instructions in the basic block. The restore instruction shown after basic block E is inserted as the last instruction in basic block E. However, the restore instruction on the control flow edge between basic block D and basic block F cannot be inserted into basic block D, because that would corrupt the value of the register in basic block E, and the restore instruction cannot be inserted into basic block F because there is an execution path that would reach that restore without any save instruction.

Sometimes the placement of spill code requires the introduction of a *jump block*. A jump block is a new basic block that is inserted into the control flow graph specifically to contain spill code that cannot be placed in other basic blocks in the control flow graph. The method for inserting a jump block on a jump edge involves changing the target of the jump instruction at the beginning of the jump edge to the beginning of the new jump block location, then inserting a new jump instruction at the end of the jump block whose target is the original destination of the jump edge.

The jump edge cost model takes into consideration the additional dynamic overhead introduced when a jump block must be inserted into the control flow graph. In the jump edge cost model, save or restore instructions that must be inserted on a jump edge are assumed to require a jump instruction to be inserted as well. The inserted jump instruction has a dynamic overhead equal to the dynamic execution count of the jump edge. For the initial save/restore sets determined using the shrink-wrapping technique, the cost of a jump instruction is divided among all the callee-saved registers that have spill locations on the corresponding jump

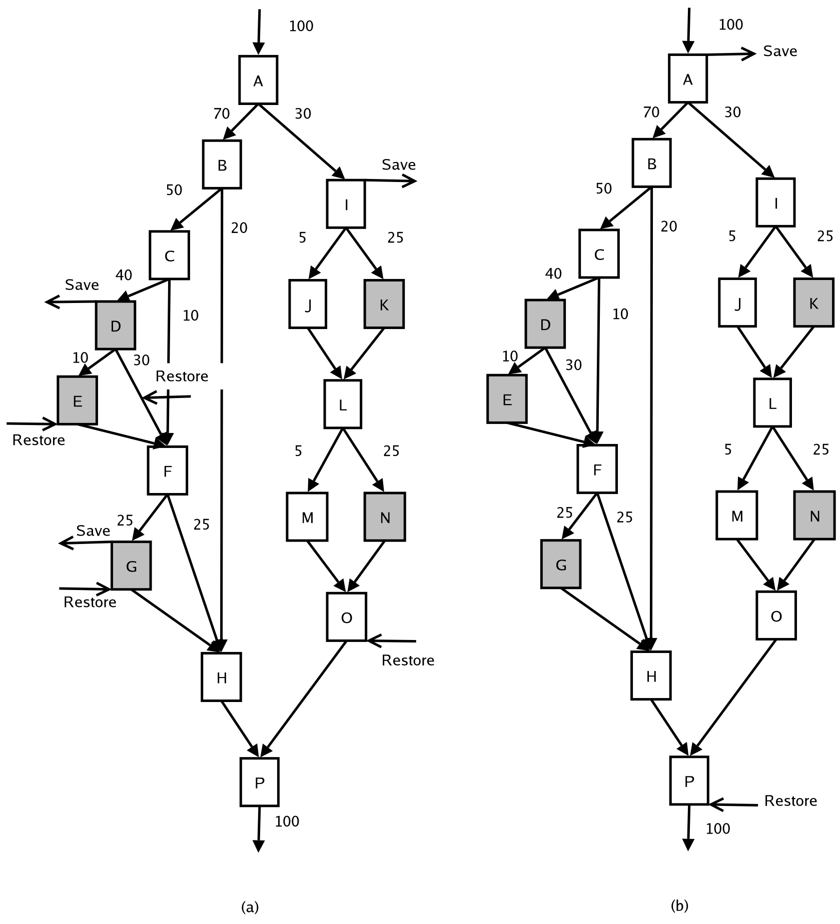

Figure 4. Example of callee-saved spill code placement computed using hierarchical spill code placement algorithm with: (a) execution count cost model, and (b) jump edge cost model.

edge. For new save/restore sets created in the topological-order traversal of the PST, each callee-saved spill instruction placed on a jump edge is assigned the complete cost of the jump instruction.

The jump edge cost model is more precise than the execution count cost model, but is not completely precise for multiple reasons. One reason is that only the first spill code instruction inserted on a jump edge requires the insertion of a jump instruction. Spill code for other callee-saved registers can be inserted in jump blocks that were previously created without having to insert additional jump instructions. This is modeled correctly with the initial placement determined using the shrink-wrapping technique, but is not precisely modeled when one or more save/restore sets are created or removed. The incremental change in jump instruction cost cannot be precisely modeled in one iteration of the hierarchical spill code placement algorithm, as the order in which callee-saved registers are evaluated may change the cost of each inserted jump instruction. The hierarchical spill code placement algorithm is limited to one iteration to avoid additional algorithmic complexity, which is discussed in detail later in the section.

A second reason the jump edge cost model lacks precision is that it is sometimes possible to change the *jump alignment* of the procedure. Jump alignment, also known as branch alignment, is a method of restructuring the control flow graph to reduce the number of jump instructions executed on the execution paths with the highest execution frequencies[9]. The jump alignment problem is beyond the scope of work presented in this paper.

To illustrate the difference between the more precise jump edge cost model and the execution count cost model in the hierarchical spill code placement algorithm, consider again the example shown in Figure 3. When analyzing Region 1, the cost of the contained save/restore set Set 1 is 110 using the jump edge cost model; 40 for the save instruction, 10 for the restore instruction in basic block E, and 60 for the restore instruction and the jump instruction required on the jump edge between D and F. The cost of Set 1 is now greater than the cost of the boundaries of Region 1, and the contained save/restore set is replaced with a new save/restore set, Set 6, at the boundaries of Region 1. The total cost of all the contained save/restore sets in Region 2 is now 150, the sum of Set 6 and Set 2. This cost is greater than the cost of the boundaries of Region 2. Therefore, the sets contained in Region 2 are deleted and a new save/restore set, Set 7, is added at the boundaries of Region 2 at a cost of 140. The results of the analysis for Region 3 are unaffected by using the jump edge cost model because none of the save or restore instructions are required to be inserted on jump edges. When analyzing Region 4, there are two contained save/restore sets. Set 7 has a cost of 140 and Set 5 has a cost of 60. Since the costs of the contained save/restore sets and the boundaries of the maximal SESE region are equal, the contained sets are removed and a new save/restore set is created at the boundaries of Region 4, which correspond to procedure entry/exit. The final placement of save and restore code is illustrated in Figure 4(b).

Chow does not allow jump blocks to be inserted in his shrink-wrapping technique, effectively assigning infinite cost to each jump edge. The jump edge cost model presented here uses a more precise measure of cost for jump edges. If the execution count of jump edges is minimized, as would be the case in a procedure where jump alignment has been performed, the jump edge cost model for spill code placed on jump edges more closely represents the real cost of inserted spill code, and the results of the hierarchical spill code placement algorithm will be affected less when the jump edge cost model is used compared to the execution count cost model.

The hierarchical spill code placement algorithm with the jump edge cost model generalizes both Chow's shrink-wrapping technique, and the procedure entry/exit placement technique. The spill code locations determined by the shrink-wrapping technique are a special case of the hierarchical spill code placement algorithm where the minimum cost locations for the placement of callee-saved save and restore instructions occur at the bottom of the hierarchical structure of the procedure. Placement of spill code at procedure entry/exit is also a special case of the hierarchical spill code placement algorithm where the minimum cost locations for the placement of callee-saved spill code occurs at the top of the procedure's hierarchical structure.

4.9. Algorithmic Complexity

In this subsection, the algorithmic complexity of the hierarchical spill code placement algorithm is analyzed.

Johnson et al. describe a fast *cycle equivalence* algorithm that allows SESE regions and the PST to be computed in linear time[8].

The algorithmic complexity of Chow's shrink-wrapping technique is not explicitly stated in prior work. For each of the n edges in a control flow graph, the number of data flow equations is bounded by a small constant R equal to the number of registers in the target processor. The number of iterations required to propagate the data flow equations through the control flow graph is bounded by the maximum distance between two control flow edges, which is n. So the the algorithmic complexity of the shrink-wrapping technique is $O(n^2)$.

The computation of save/restore sets is an iterative data flow analysis quite similar to shrink-wrapping, and the complexity of the computation is also $O(n^2)$.

The cost function used in the hierarchical spill code placement algorithm is computed in constant time, as are the creation and removal of save/restore sets. The propagation of save/restore set changes upward through the hierarchy of the PST is done in linear time. Since the topological-order traversal of the PST is bounded by the number of edges in the control flow graph, the analysis of all save/restore sets in the PST is computed in $O(n^2)$ time, and that portion of the algorithm is repeated the small constant R times.

The complexity of the entire hierarchical spill code placement algorithm is $O(n^2)$, which is equivalent to the complexity of the shrink-wrapping technique.

5. Experimental Results

The hierarchical callee-saved spill code placement algorithm with the jump edge cost model described in the previous section was implemented in the Gnu Compiler Collection (GCC) and applied to the SPEC CPU2000 integer benchmark programs.[1] The execution count cost model is not evaluated experimentally because it generates code that, without modification, cannot always be executed. This is due to spill instructions being placed on jump edges which have no physical memory allocated to them.

The register allocator of GCC was replaced with a Chaitin/Briggs style graph-coloring register allocator. This was done to ensure that identical, optimized register allocations and register assignments were used for each post register allocation spill code placement technique. GCC's register allocator was not used because it relies heavily on a post register allocation reload module that can alter a register assignment and introduce additional spill code. Callee-saved spill code placement was performed after register allocation. The target microprocessor is a PA-RISC with 24 general purpose registers available for allocation. There are

[1]The one integer benchmark program written in C++ was excluded due to lack of operating system library support at the time of the experiment.

13 callee-saved registers in the register usage convention used in these experiments.

To accurately observe the contribution of the register allocator and the hierarchical spill code placement algorithm without the effects of other compiler optimizations, only dynamic spill code overhead is measured. Figure 5 shows the total dynamic spill code overhead for each of the SPEC CPU2000 integer benchmarks. The totals include all load and store instructions generated by the register allocator as well as callee-saved save and restore instructions. Each of the three data sets have the exact same register allocation, with the only differences being the placement of callee-saved save/restore instructions inserted by the hierarchical spill code placement algorithm. The data set labeled Optimized corresponds to the hierarchical callee-saved spill code placement algorithm, the data set labeled Shrinkwrap corresponds to an implementation of Chow's shrink-wrapping technique, and the data set labeled Baseline corresponds to the placement of save/restore instructions at procedure entry and exit.

These results show that the hierarchical spill code placement algorithm nearly always generates less dynamic overhead than both the shrink-wrapping technique and the procedure entry/exit placement technique. In no case does the hierarchical spill code placement algorithm generate more dynamic overhead than either of the other placement techniques. One benchmark, mcf, has very small spill code overhead and is not visible in Figure 5. The procedures in mcf are relatively small compared to procedures in the other benchmark programs. The graph-coloring register allocator is often able to perform a register allocation that uses only the caller-saved registers with no spilling required. For the few procedures in mcf that do utilize callee-saved registers, the dynamic execution count is very small compared to procedures in the other benchmark programs.

Figure 5 also shows three benchmarks; gzip, bzip2 and twolf, where the dynamic spill code overhead produced by the placement of save/restore instructions using the shrink-wrapping technique is greater than the dynamic spill code overhead produced by the procedure entry/exit placement technique. For each of those three benchmarks, the hierarchical spill code placement algorithm generates less dynamic spill code overhead than the procedure entry/exit placement technique. In particular, for benchmark gzip, the hierarchical spill code placement algorithm produces 17% less dynamic spill code overhead than the procedure entry/exit placement technique.

Table 1 shows the ratios of dynamic spill code overhead for all of the benchmark applications. The ratios in Table 1 are computed by dividing the total dynamic spill code overhead from the hierarchical spill code placement algorithm, and the total amount of dynamic spill code overhead from the shrink-wrapping technique, by the total dynamic spill code overhead generated by the procedure entry/exit spill code placement technique. On average, the hierarchical spill code placement algorithm generates 15% less dynamic spill code overhead compared to the procedure entry/exit placement technique. In contrast, the amount of dynamic spill code overhead generated when using the shrink-wrapping technique is reduced by less than 1% on average compared to the procedure entry/exit spill code placement technique. The results of the hierarchical spill code placement algorithm are sometimes much better than the average indicates. For example, in the gcc benchmark program, the hierarchical spill code placement algorithm reduces the amount of dynamic spill code overhead by greater than 40%. The gcc benchmark program is the largest program in the SPEC CPU2000 integer benchmark suite, comprising nearly a third of all the procedures in the benchmarks. In contrast, the shrink-wrapping technique reduces dynamic spill code overhead in gcc by only 6% compared to the procedure entry/exit placement technique. The crafty benchmark has a greater than 50% reduction in dynamic spill code overhead when using the hierarchical spill code placement algorithm compared to 6% when using the shrink-wrapping technique. Both the gcc and crafty benchmarks utilize a number of unconditional jump instructions (gotos), which tend to increase the number of jump edges that can be exploited with the jump edge cost model but are ignored in the shrink-wrapping technique.

Compile times were measured for both the hierarchical spill code placement algorithm and shrink-wrapping technique. Each technique was implemented using similar dynamic memory allocation techniques. All experiments were run with an optimization level of O2 on a HP C3000 workstation with 1.5GB of RAM. Table 2 shows the incremental increase in compilation time of both the shrink-wrapping technique and the hierarchical spill code placement algorithm compared to compile times of the entry/exit placement technique. The results show that the additional compilation time required for the hierarchical spill code placement algorithm is within an average factor of 6 compared to the additional compilation time required for shrink-wrapping. Additional compilation time is expected because the shrink-wrapping technique is a component of the hierarchical spill code placement algorithm.

6. Summary

This paper presents a new, post register allocation approach to placing spill code. The hierarchical spill code placement algorithm generalizes prior techniques for placing callee-saved save and restore instructions, and produces a spill code placement with minimized dynamic execution count. The post register allocation spill code placement problem is solved optimally using the execution count cost

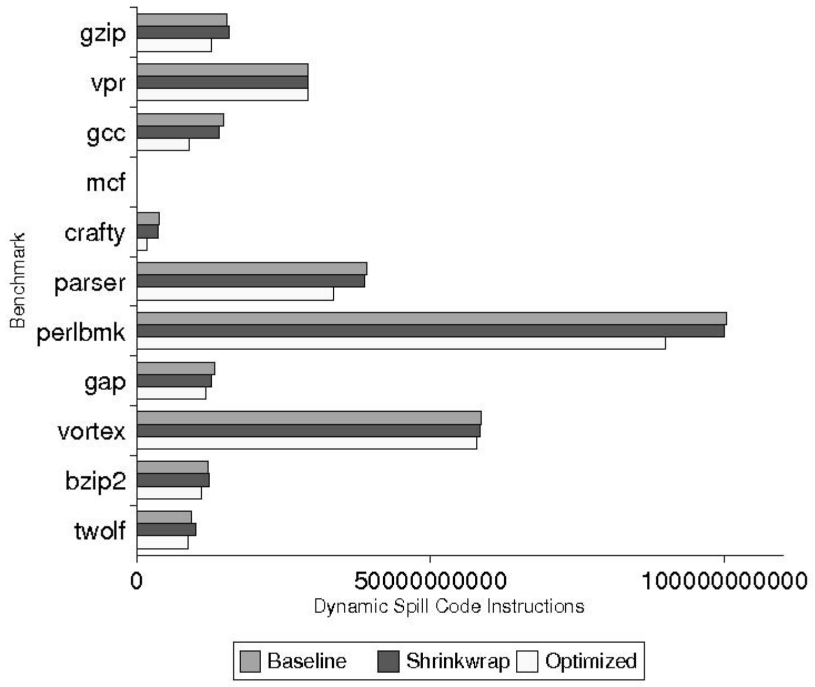

Figure 5. Spill code overhead generated for SPEC CPU2000 integer benchmarks with callee-saved save/restore placement determined by; optimized hierarchical algorithm, shrink-wrapping, and procedure entry/exit.

model in the hierarchical spill code placement algorithm, while a more precise jump edge cost model allows placement of spill code on jump edges where previous techniques do not. Algorithmic complexity of the hierarchical spill code placement algorithm is equivalent to that of the best previous technique. The hierarchical spill code placement algorithm is implemented in the GCC compiler with a graph-coloring register allocator and applied to the SPEC CPU2000 integer benchmark applications. Results show that average dynamic spill code overhead is reduced by 15% compared to previous spill code placement techniques.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[3] P. Briggs, K. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[4] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 192–203, June 1991.

[5] G. Chaitin. Register allocation and spilling via graph coloring. In *Symposium on Compiler Construction*, volume 17, pages 98–105, 1982.

[6] F. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of Conference on Programming Language Design and Implementation*, June 1988.

[7] F. Chow and J. Hennessey. The priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems*, 12(4):501–536, October 1990.

[8] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–185, June 1994.

[9] S. McFarling and J. Hennesey. Reducing the cost of branches. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 396–403, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[10] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[11] M. D. Smith, N. Ramsey, and G. Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM Press.

SPEC CPU2000 Integer Benchmark	Optimized Placement / Baseline Placement	Shrinkwrap Placement / Baseline Placement
gzip	83.0%	102.6%
vpr	99.5%	100.0%
gcc	59.6%	93.9%
mcf	100.0%	100.0%
crafty	44.0%	93.3%
parser	85.8%	99.0%
perlbmk	89.7%	99.6%
gap	88.5%	95.4%
vortex	98.8%	100.0%
bzip2	90.2%	100.5%
twolf	93.9%	108.0%
Average	**84.8%**	**99.3%**

Table 1. Ratios of dynamic spill code overhead for optimized spill code placement compared to shrink-wrapping relative to saving/restoring at procedure entry and exit.

SPEC CPU2000 Integer Benchmark	Increcmental Compile Time / for Shrink-wrapping (sec)	Incremental Compile Time for Optimized Placement (sec)	Ratio
gzip	0.42	2.2	5.24
vpr	0.59	4.74	8.03
gcc	115.10	269.02	2.34
mcf	0.05	0.24	4.8
crafty	0.34	1.15	3.38
parser	1.04	8.40	8.08
perlbmk	15.8	62.99	3.99
gap	10.51	64.67	6.15
vortex	5.23	40.68	7.78
bzip2	0.50	3.70	7.40
twolf	2.88	7.58	2.63
Average	**13.86**	**42.30**	**5.44**

Table 2. Ratios of incremental compilation times for shrink-wrapping and optimized spill code placement compared to entry/exit placement compilation times.